Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

The lifecycle of Technical Debt that manifests in both source code and issue trackers

Jie Tan [a], Daniel Feitosa [b,*], Paris Avgeriou [b]

[a] Academy of Military Science, Beijing, China
[b] Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, The Netherlands

ARTICLE INFO

Keywords:
Technical Debt
Source code
Issue tracker

ABSTRACT

Context: Although Technical Debt (TD) has increasingly gained attention in recent years, most studies exploring TD are based on a single source (e.g., source code, code comments or issue trackers).
Objective: Investigating information combined from different sources may yield insight that is more than the sum of its parts. In particular, we argue that exploring how TD items are managed in both issue trackers and software repositories (including source code and commit messages) can shed some light on what happens between the commits that incur TD and those that pay it back.
Method: To this end, we randomly selected 3,000 issues from the trackers of five projects, manually analyzed 300 issues that contained TD information, and identified and investigated the lifecycle of 312 TD items.
Results: The results indicate that most of the TD items marked as resolved in issue trackers are also paid back in source code, although many are not discussed after being identified in the issue tracker. Test Debt items are the least likely to be paid back in source code. We also learned that although TD items may be resolved a few days after being identified, it often takes a long time to be identified (around one year). In general, time is reduced if the same developer is involved in consecutive moments (i.e., introduction, identification, repayment decision-making and remediation), but whether the developer who paid back the item is involved in discussing the TD item does not seem to affect how quickly it is resolved.
Conclusions: Investigating how developers manage TD across both source code repositories and issue trackers can lead to a more comprehensive oversight of this activity and support efforts to shorten the lifecycle of undesirable debt.

1. Introduction

The Technical Debt (TD) metaphor refers to the trade-off between the short-term benefits of “cutting corners” in software development and the long-term sustainability of the software being developed [1]. TD can manifest in different artifacts, from requirements to architecture and from source code to documentation [1]; in this study we focus on TD that manifests in source code. While source code TD is ultimately paid back also in source code [2,3], it is often recorded in other sources as well; for example, it may be identified and discussed in issue trackers [4], while its repayment may be additionally recorded in the commit messages of version control systems [5].

Most of the research work on source code TD has focused solely on a single source (e.g., source code comments or issues), while few studies have looked at a combination of sources [6–8]. However, to the best of our knowledge, none of the existing studies has combined different sources to study the complete lifecycle of TD items, from their introduction to their repayment. This one-sided perspective often results in misleading findings and an incomplete understanding of this lifecycle; to explain this problem in more detail, we provide two examples.

First, while previous studies have shown that the repayment of a large percentage of TD is documented in issue trackers [4,8], it is not always true that the TD is paid back. For example, if the remediation of a TD item is added to a patch, and the patch is rejected, the item is not actually fixed in the project. Thus, the issue has been seemingly resolved in the issue tracker, while the code still remains unchanged. In such cases, it is unclear whether it was unnecessary to fix the TD item in the code or there was a decision to keep the code unchanged.

Second, if one focuses only on TD in source code comments (so-called Self-Admitted Technical Debt), one may also get false positives [5,9]. For example, the deletion of comments does not necessarily imply the remediation of the debt. Also, comments can be removed accidentally when entire classes or methods are deleted [6]. Finally,

* Corresponding author.
E-mail addresses: j.tanjie@outlook.com (J. Tan), d.feitosa@rug.nl (D. Feitosa), p.avgeriou@rug.nl (P. Avgeriou).

https://doi.org/10.1016/j.infsof.2023.107216
Received 4 July 2022; Received in revised form 13 February 2023; Accepted 31 March 2023
Available online 5 April 2023

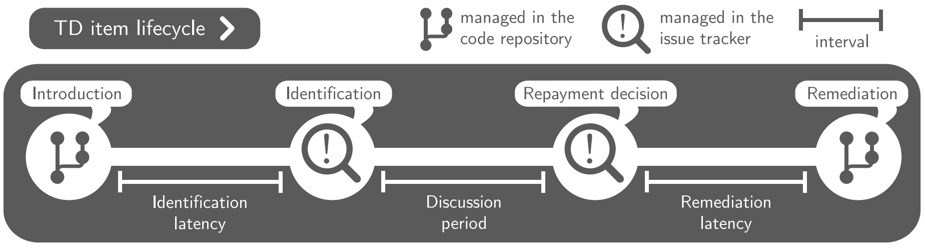


Fig. 1. Lifecycle of TD items that manifest in both issue trackers and source code.

remediation can also happen as a ripple effect of maintenance, e.g. as part of a bug fix, instead of directly intending to rectify sub-optimal code. By just looking at the code and accompanying comments, we are often unaware of developers' intent and to what extent and how they prioritize TD remediation.

To address such incorrect interpretations of the lifecycle of TD items, we combine information from two sources: source code repositories and issue trackers. Specifically, we consider that this lifecycle begins with TD items incurred in source code, then reported and discussed in issue trackers and finally paid back in source code (for more details, see Section 2). To study this lifecycle, we examine the percentage of issues resolved in issues trackers also being resolved in source code, the length of the intervals between the various moments in the lifecycle, and how the number and involvement of developers are related to the lifecycle.

The results show that most (but not all) TD items identified and mentioned as resolved in issue trackers are also paid back in source code, while only about one-third are discussed amongst developers in the issue tracker after being identified. TD items commonly stay dormant for a long time after being introduced, but they do get resolved within only a few days after their identification. Some types of debt (e.g., Design Debt) are more likely to be paid back than others (e.g., Test Debt). Finally, the involvement of the same developer in introduction and identification, or identification and repayment tends to lead to quicker turnarounds (i.e., faster identification or repayment respectively). However, we did not observe any significant changes to the length of the discussion period associated with the number or kind of developers involved in the conversation.

Investigating how developers manage TD across both source code repositories and issue trackers can lead to a more comprehensive oversight of this activity and support efforts to shorten the lifecycle of undesirable debt. For example, learning that TD items commonly stay dormant for most of their lifecycle is an indicator that more research into early TD detection and reporting is needed, and that practitioners are advised to (continue to) report it and encourage others to do the same. Also, not many of the reported and successfully resolved TD items are discussed in issue trackers. However, as these TD items survive longer than other TD items in average, further research can help in expediting the remediation of these items; one way is to mine discussions from other issues addressing similar TD-related topics, and subsequently provide suggestions for repayment.

The remaining sections are organized as follows. The paper starts by characterizing the lifecycle of TD items (as considered in this work) in Section 2. The study objectives, the research questions, and details regarding the data collection and analysis are reported in Section 3. Section 4 presents the results of our study and Section 5 interprets the results, discusses their implications, and reports the threats to the validity of our study. Section 6 summarizes related work and further contextualizes our study. Our final thoughts are shared in Section 7.

2. Technical debt item lifecycle

To keep TD under control, practitioners may adopt different strategies [10,11], and apply various frameworks especially designed for this purpose [12–14]. These approaches support a number of Technical Debt Management (TDM) activities, which range from the identification of an item to its remediation [1]. This range represents the lifecycle of a TD item, spanning the time period from the introduction of the debt to its remediation.

The lifecycle of a TD item may be modeled in various ways depending on the sources (i.e., the information taken into account) [15–17]. For example, it may span from the moment an item is self-admitted in a code comment, to the moment the comment is removed. Or, it may span from the moment it is documented in a backlog, creating a ticket, to the moment the ticket is resolved. In this study, we consider a typical lifecycle for TD items that manifest in both the source code and issue trackers, as depicted in Fig. 1. It starts with the introduction of TD and continues with three TDM activities: identification, assessment and remediation [1,18]. The white circles represent the four key moments in the lifecycle of each TD item, while the icons in those circles indicate the data source where TD is managed during these moments, i.e., source code repository or issue tracker. The spaces between the circles represent intervals between the four moments in the TD item lifecycle.

TD introduction and *remediation* are the moments when TD items are introduced and paid back in source code, respectively [1]. Both actions are represented by a change to the source code base, e.g., a commit to a Git[1] repository. A change can lead to the introduction of TD if e.g. the new code contains a workaround, or a new component (e.g., a function or class) is added without proper documentation or unit tests [2]. Conversely, a commit that removes the workaround or adds the missing elements would pay back the debt. We note that the introduction of TD can be deliberate or inadvertent and making such a distinction is out of the scope of this study (see Section 3.1).

TD identification refers to the moment when TD items are first identified in issue trackers (e.g., Jira[2] or Github issues[3]) [1]. Issues are created independently of the codebase evolution and, thus, can be reported by those who introduced the TD item or others that noticed it. Moreover, issue trackers offer several features to allow monitoring the TD item, e.g., denoting issue status as 'open', 'closed' and 'resolved', adding user comments to enable discussions, and smart links to codebase elements such as individual commits and files to point out problems and solutions. Typically, after identifying TD items, developers have several discussions to determine if and how to pay them back in source code. When such discussions conclude, *TD repayment decision* refers to the moment when developers have determined how to pay back the TD and subsequently mentioned their decision in the issue tracker [19].

Regarding the intervals, *identification latency* refers to how long it takes to identify (and thus document in an issue) the technical debt after introducing it in source code. Next, the *discussion period* refers to how long developers take to discuss and decide how to resolve the technical debt after being identified in the issue tracker. Finally,

[1] https://git-scm.com/.
[2] https://www.atlassian.com/software/jira.
[3] https://docs.github.com/en/issues.

with a decision made, developers take action and refactor the code accordingly. In the best case scenario, developers will apply changes in a short time. However, developers may also resolve it later in some cases, e.g., when the TD item is not urgent or needs to be resolved by a specific developer. Thus, *remediation latency* refers to how long developers take to pay back technical debt in source code after a solution is decided.

3. Study design

In this work, we designed a case study since we seek to investigate a phenomenon in its natural settings [20], i.e., the lifecycle of technical debt items that are managed in both issue trackers and software repositories. As is typical for case studies, we employ multiple methods of data collection to gather information from a variety of sources. Moreover, we consider a quantitative approach as the research process since we use metrics and statistical analysis. The case study was designed based on the guidelines of Runeson et al. [21] and is reported according to the Linear Analytic Structure [21].

3.1. Objectives and research questions

The goal of our study, described according to the Goal-Question-Metrics (GQM) approach [22], is to "*analyze* software repositories *for the purpose of* investigating the lifecycle of technical debt items *with respect to* the consistency between the repayment decision and remediation moments, the length between the various moments in the lifecycle, and how the number and involvement of developers are related to the lifecycle *from the point of view of* software developers *in the context of* open source software". This objective is further refined in terms of the following research questions.

RQ1: To what extent technical debt items identified and tagged as 'resolved' in issue trackers are actually paid back in source code?

Specifically we focus on investigating to what extent TD items that are tagged as 'resolved' in issues can be mapped to the commits that address them. To this end, we first match each 'resolved' issue with the corresponding commit(s), by looking at the messages in the commits. Then, we manually check if TD-related problems reported in the issue documentation (i.e., summary, description, and comments) are actually resolved in the associated commits (for details, see Sections 3.3.4 and 3.3.5). We clarify that we do not make a distinction between TD that is introduced deliberately or inadvertently since it would require asking the developers (see also RQ2); a TD item that is only later reported in an issue tracker, can be incurred both deliberately and inadvertently. Investigating this research question provides an insight into the likelihood of the TD-related issues marked as resolved in issue trackers actually being repaid in the code. This gives a more accurate view of TD remediation and can help software engineers to further monitor TD management activities and review issue documentation practices. It can also help researchers design and refine tools to support TD monitoring in issue trackers.

RQ2: How long are the intervals between introducing, identifying, deciding to repay and actually repaying TD items?

This research question focuses on investigating the intervals between the various moments in a TD item lifecycle. Specifically, we consider the three intervals depicted in Fig. 1, i.e., *identification latency*, *discussion period* and *remediation latency*. Such an investigation provides a more complete picture of how and when developers manage technical debt items by analyzing their introduction and remediation in the source code, together with their identification, discussion and repayment decision-making in the issue trackers. Moreover, software engineers and project managers can use this information by paying more attention to intervals that grow relatively long.

We note that by looking into *identification latency*, we do not imply that the underlying TD item is introduced inadvertently. Without developers' involvement in the study, we cannot make an accurate distinction. For example, the introduction may be deliberate but then only reported as an issue later for various reasons (e.g., the consequences of the corner-cutting grew, or the developer simply decided to do so later).

RQ3: How is developer involvement associated with the length of each interval?

> **RQ3.1:** Does the *identification latency* change when the identifying developer also introduced the TD item?
> **RQ3.2:** Does the *discussion period* change according to the number of participating developers?
> **RQ3.3:** Does the *remediation latency* change when the decision-making developer or the identifying developer also paid back the TD item?

In this research question, we seek to identify how particular development team dynamics may influence the length of the three investigated intervals. In particular, we explore if the duration of an interval changes when the same developer is involved in two distinct moments (i.e., introduction and identification for RQ3.1, and repayment decision-making and remediation, or identification and remediation for RQ3.3). We also explore if the number of developers participating in the discussion of a TD item is related to its duration (RQ3.2). Such findings can help developers understand how to more effectively participate in TD management and particularly improve the efficiency of TD remediation. For example, if more developers participating in a tracked issue can speed up the decision of TD remediation, then project managers can identify potentially relevant developers and encourage them to engage in the discussions.

3.2. Projects for data extraction

To perform this study, we focused on the Apache Software Foundation (ASF), given the reputation of Apache projects for high quality and long-term stability.[4] Apache projects are developed and maintained by technical experts following Apache-wide meritocratic rules. In addition, the ASF is the world's largest open source foundation and contains more than 300 top-level projects[5]; consequently it covers a wide range of languages, sizes and number of contributors. Furthermore, Apache projects are commonly selected in technical debt studies (see e.g. [4,9,23]). The project selection process followed five inclusion criteria:

1. The project must be publicly available (including both the repository and the issue tracker), and it must also use Git as the version control system and Jira[6] as the issue tracker. We selected Git and Jira since these are well-established and popular tools among practitioners and allow us to standardize the data collection and analysis.
2. The project must be under development for at least five years and have more than 1000 commits. A long history enables developers to observe the consequences of TD and fix it. Moreover, this number of revisions provides a sufficient set of repeated measures for statistical analysis [3].
3. The project must have at least 150,000 source lines of code (SLOC), and 5000 issues in the issue tracker. Larger projects typically cover more TD types, and enough issues indicate that developers are using the issue tracker frequently and actively.
4. The project must involve at least 100 developers. This ensures some diversity among developers (e.g., seniority level).

[4] http://www.apache.org/foundation/how-it-works.html, visited February 2021.
[5] https://www.apache.org/index.html#projects-list, visited February 2021.
[6] https://jira.apache.org/.

Table 1
Details of the five selected projects.

	Camel	Hadoop	Hbase	Impala	Thrift
SLOC[a]	1580k	1858k	839k	607k	152k
Source files[a]	20,252	11,602	4800	3039	741
Commits	50,380	24,612	18,340	9661	6197
PMC members[b]	43	124	58	37	21
Committers[b]	85	242	101	65	41
Developers[b]	722	22,954	2992	149	353
Issues	14,411	16,808	24,342	9733	5196
Languages	Java	Java	Java	C++, Java, and Python	C++, Java, and C
Categories[c]	Integration, cloud, java, library	Big-data, hadoop, cloud	Database	Hadoop, sql	http, library, network-client, network-server

[a]Of the main languages only.
[b]Roles clarified at https://www.apache.org/foundation/how-it-works.html#roles.
[c]Based on Apache's categories (https://projects.apache.org/projects.html?category).

5. The project must require developers to include Jira ticket numbers in the commit messages when providing code patches (e.g., Apache Camel[7]). This is necessary to support linking the issues (in the issue tracker) to the commits (in the source code repository).

To enhance the representativeness of the sample (and mitigate threats to external validity), we randomly selected five projects out of over 200 projects that fulfill the aforementioned criteria. Table 1 presents the details of each project, including information on the domain (i.e., categories), source code (i.e., main languages, files and SLOC), contributors (i.e., developers, committers and Project Management Committee—PMC—members) and issues. We analyzed the latest released versions on February, 2021.

Looking at the management structure of the projects, all follow ASF's contribution guidelines.[8] ASF adopts a role-based structure rather than a hierarchical one but projects have the freedom to further organize themselves. We inspected the contribution guidelines of each project and found no evidence of any deviation from the following default structure. ASF defines roles that have different rights and responsibilities in the project.

A **developer** (or contributor) is a person who contributes to the codebase or discussions (incl. reporting issues). Code changes are based on patches, which are commonly submitted as pull requests (PRs). We found a few PRs that contain multiple commits to address a single issue, but these are exceptions in the studied projects. Also single-commit PRs seem a common practice, as supported by the Thrift guidelines[9]: "All pull requests should contain a single commit per issue".

Committers are developers with repository write permission. We note that non-committer developers still appear as authors in repository commits. In practice, they allow short-term decisions since they signed a Contributor License Agreement.[10]

Neither developers nor committers make decisions, e.g., of what changes are accepted into the codebase. These decisions lie with **PMC members**. They are elected based on merit and, among other rights, can propose the promotion of developers to committers. The PMC controls the project (incl. acceptance of fixes to issues) via a lazy consensus approach,[11] i.e., a few positive votes with no negative votes are enough to apply a change.

3.3. Variables and data collection

This section presents the set of variables necessary to answer the research questions and the major steps of the data collection process. In particular, each unit of analysis represents the lifecycle of a single TD item, which encompasses: (a) the commit(s) that *introduced* the debt; (b) the issue documenting the *identification* and *discussion* of the debt item; and (c) the commit(s) that *pay back* the debt. The issue (b) includes the *issue key*, *issue sections* (i.e., summary, description and comments), *resolution status*, *developers* who reported, discussed and resolved the issues, *creation timestamp*, *resolution timestamp* and the *types of technical debt*. The commits (a) and (c) correspond to the analyzed issue, and include *repository name*, *commit hashcode*, *commit date*, *author's name*, *author's email* and *commit message*. We note that, for privacy reasons, the names and emails of developers were anonymized in the extraction and not stored at any moment.

Data collection is comprised of four main steps, as shown in Fig. 2: (i) extracting and filtering issues, (ii) manually identifying and classifying technical debt, (iii) matching the analyzed issues to the commit(s) that pay back the TD item, and (iv) determining the commits that introduced the identified TD item. These steps are elaborated in the following paragraphs.

3.3.1. Step I: Selecting and filtering issues

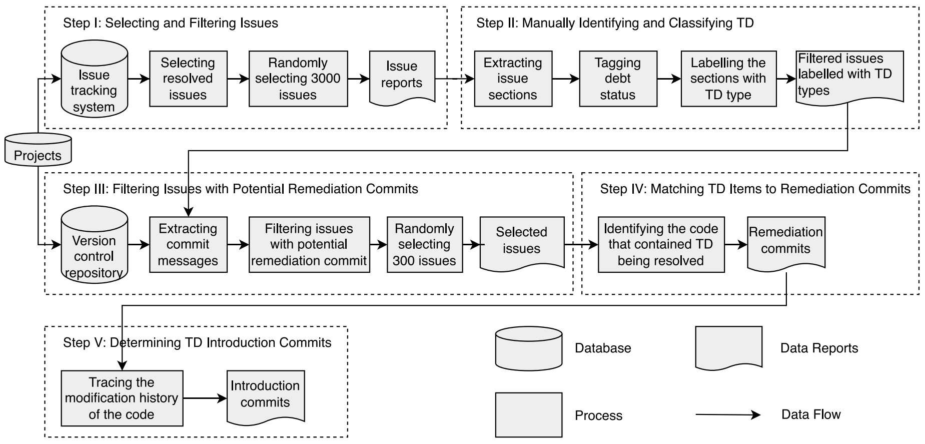
In order to investigate TD items that exist in both source code repository and issue trackers, we begin by collecting all issue reports (i.e., metadata and issue sections) and selecting those that are resolved, i.e., marked with status 'resolved' or 'closed'. This is because all resolved issues are potential candidates for containing TD items. To confirm whether they indeed comprise TD, we need to manually analyze them in the following step. Since manual analysis of the issues is extremely time-consuming, we can only analyze a subset. Thus, we randomly select 600 issues from each project, totaling 3000 issues from the five projects.

3.3.2. Step II: Manually identifying and classifying TD

Similarly to Li et al. [4], we extract the sections of tracked issues (i.e., summaries, description, and comments) and analyze them at the sentence level. In particular, we search for sentences that indicate technical debt, and tag the debt status in each one (i.e., paid back or not). To classify TD, we use the classification framework from Li et al. [4], who also used it to classify the types of TD contained in issues; the framework itself, is based on a well-known TD classification framework provided by Alves et al. [24]. We label the TD sections according to these types. The dataset with the classification information for the 3000 issues is available in our replication package [25].

Given the labor-intensive nature of the analysis, we want to augment the chances of finding relevant data (i.e., related to code changes). Thus, we discard sections from the classification framework from Li et al. [4] that are not closely related to the source code (e.g., classified

[7] https://camel.apache.org/manual/latest/contributing.html, visited February 2021.
[8] https://www.apache.org/foundation/how-it-works.html.
[9] https://thrift.apache.org/docs/HowToContribute.
[10] https://www.apache.org/foundation/how-it-works.html#committers.
[11] https://www.apache.org/foundation/how-it-works.html#decision-making.



Step I: Selecting and Filtering Issues
Issue tracking system
Selecting resolved issues
Randomly selecting 3000 issues
Issue reports
Step II: Manually Identifying and Classifying TD
Extracting issue sections
Tagging debt status
Labelling the sections with TD type
Filtered issues labelled with TD types
Projects
Step III: Filtering Issues with Potential Remediation Commits
Version control repository
Extracting commit messages
Filtering issues with potential remediation commit
Randomly selecting 300 issues
Selected issues
Step IV: Matching TD Items to Remediation Commits
Identifying the code that contained TD being resolved
Remediation commits
Step V: Determining TD Introduction Commits
Tracing the modification history of the code
Introduction commits
Database
Data Reports
Process
Data Flow

Fig. 2. An overview of the data collection.

Table 2
TD types that are detected in issue trackers.

Types	Indicator	Definition
Architecture debt	Violation of modularity (VioMod)	Because shortcuts were taken, multiple modules became inter-dependent, while they should be independent.
	Using obsolete technology (ObsTech)	Architecturally-significant technology has become obsolete.
Code debt	Complex code (CpxCd)	Code has accidental complexity and requires extra refactoring action to reduce this complexity.
	Dead code (DedCd)	Code is no longer used and needs to be removed.
	Duplicated code (DupCd)	Code that occurs more than once instead of as a single reusable function.
	Low-quality code (LQualCd)	Code quality is low, for example because it is unreadable, inconsistent, or violating coding conventions.
	Multi-thread incorrectness (MTCor)	Thread-safe code is not correct and may potentially result in synchronization problems or efficiency problems.
	Slow algorithm (SlAlg)	A non-optimal algorithm is utilized that runs slowly.
Defect debt	Uncorrected known defects (Def)	Defects are found by developers but ignored or deferred to be fixed.
Design debt	Non-optimal defects (Des)	Non-optimal design decisions are adopted.
Test Debt	Expensive tests (ExpTst)	Tests are expensive, resulting in slowing down testing activities. Extra refactoring actions are needed to simplify tests.
	Flaky tests (FlaskTst)	Tests fail or pass intermittently for the same configuration.
	Lack of tests (LacTst)	A function is added, but no tests are added to cover the new function.
	Low coverage (LCvg)	Only part of the source code is executed during testing.

Table 3
Number of issues per TD type.

Debt type	Number of issues											
	Camel		Hadoop		Hbase		Impala		Thrift		Sum	
Architecture	10	(6)	12	(6)	6	(7)	5	(2)	9	(9)	42	(30)
Code	52	(10)	122	(90)	109	(50)	55	(26)	130	(34)	468	(210)
Defect	1	(1)	2	(4)	0	(1)	0	(1)	1	(2)	4	(9)
Design	61	(17)	35	(73)	82	(50)	73	(43)	74	(27)	325	(210)
Test	25	(7)	36	(26)	46	(15)	25	(7)	28	(8)	160	(63)
Total	149	(41)	207	(199)	243	(123)	158	(79)	242	(80)	999	(522)
Selected	61		63		63		52		61		300	

as Requirement Debt). The definitions of the different selected types and indicators of TD are shown in Table 2.

Table 1 also reports the filtered issue details for each project. We first present the number of issues that contained TD (see column 'TD Issues'). We note that in an issue where TD is discussed, other (non-TD) problems may also be addressed. Thus, we also present the number of sections that contained TD and its percentage compared to the total amount of sections in the filtered issues (see column 'TD Sections').

3.3.3. Step III: Filtering issues with potential remediation commits

First, we clone the *git* repository of each project, and use *git log* to extract the code commits together with the corresponding commit messages. According to the project selection criteria (see Section 3.2), all five projects require developers to add the Jira issue key when providing code patches. Thus, we expect the commit messages to contain the issue ID, indicating the issue that each commit addresses; taking advantage of this, we use the regular expression-based approach provided by Fischer et al. [26] to match the issue key in the commit message and thus extract those messages, whose corresponding issues are labeled as any type of TD.

Table 3 shows the number of issues per TD type. We present two numbers per project and TD type. On the left of each column is the number of issues with ID appearing in commit messages, and on the right (between parentheses) is the number of remaining issues. Overall, we were not able to find the ID for 34% of the issues and cannot determine whether and when they were resolved in the source code. Also, since there are only four Defect Debt items in total, we will not consider that debt type. Thus, the remaining 65% of the issues are relevant for this study.

Manually reviewing the code that resolved the TD item and further tracing back to the code that introduced the same item is fairly time-consuming. Thus, we decided to limit the analysis to 300 issues spread over the five projects. By investigating the total number of issues per project (second to last row of Table 3), these are unequally distributed across the projects. Thus, we randomly selected issues by using *stratified random sampling* [27], which is used to estimate population parameters efficiently when subpopulations have substantial variability [28]. For each project, we randomly selected a number of issues based on the total number of issues it contains (see Table 1).

Developers can identify TD in issue trackers in three cases: before creating an issue, during code review and after a patch is committed [4]. Since this study considers a lifecycle starting with the TD introduction in the source code, we only consider the first case (i.e., TD is the reason for creating the issue), and manually filter out the others. Based on the above criteria, we finally selected 61, 63, 63, 52, 61 issues respectively for the five projects (as shown in the last row of Table 3).

3.3.4. Step IV: Matching TD items to remediation commits

For each issue that has at least one commit mentioning its key in the message, we extract the TD items discussed in it. The TD items are identified based on context. We note that we consider the entire issue for analyzing the context, including parts that were not classified as TD. If the discussion revolves around the same problem, then one TD item is extracted. This is the common case since issues are naturally meant to discuss individual problems. In the rare cases where a different (TD-related) problem is brought as a consequence of the discussion, multiple TD items are extracted.

Once those TD items are identified, we investigated whether each TD item was actually resolved in the codebase. For that, we start by extending the set of related commits. From each commit that contains the issue key, we also get the commits in the pull request that the original one is part of. Then, with this set of potentially fixing commits, we inspect all changes at the code level to identify if the problem reported by the item is solved. If a solution had been proposed or directly reported in the issue sections, our search would start from this solution.

Finally, if we identify a fix, the respective commit is recorded. When multiple commits contribute to the fix, all are recorded. We found only seven instances of such case. The details of the selected issues, including the classification of the contained technical debt and the hashcode of the corresponding commit resolving the issue are available in our replication package [25].

3.3.5. Step V: Determining TD introduction commits

For the TD items that we established being fixed, we then proceed to identify the commit(s) that introduced the problem in the first place. Based on the identified commit at the code level, we use *git blame* to trace the modification history of these pieces of code and record the commit(s) that created those pieces of code.

Here we distinguish between introducing debt and accumulating debt. The former refers to commits that comprise a patch that passed quality control, but introduced a TD item; the latter refers to later commits that worsen or spread the problem of an existing TD item. A simple example in the dataset is "*THRIFT-2768|0*",[12] in which spacing was inconsistent (and fixed). In this case, we only record the first commit containing inconsistent spacing. In the case where we deem multiple commits to comprise a patch (e.g., "*IMPALA-5273|0*"), all were recorded.

3.4. Data analysis

For **RQ1**, to investigate to what extent technical debt items resolved in issue trackers are actually paid back in source code, we use the data collected as described in Section 3.3: specifically we look at the selected issues that contain TD items and the corresponding commits. Subsequently, we manually check if TD-related problems in the issue documentation (i.e., summary, description, and comments) are resolved in the corresponding commits.

To answer **RQ2**, we need to determine the four moments in the TD item lifecycle and then calculate the three time intervals between them (see Fig. 1). Regarding the moment of TD introduction, we first use *git blame* to trace the first commit in which developers introduced the pieces of code incurring TD. The commit date of that commit is the TD introduction moment. Regarding the TD identification moment in issue trackers, we use the date in the issue documentation where TD-related problems are first mentioned. Consequently, *identification latency* is the time difference between the moments of TD identification and TD introduction.

After TD-related problems are identified and flagged in issue trackers, developers may have several discussions about TD before they decide to resolve it. To represent the moment of TD repayment decision, we use the date in the issue documentation when developers mentioned their final decision about the remediation of the TD items. Consequently, the *discussion period* is the time difference between the moments of repayment decision and identification.

After developers decide how to resolve TD in issue trackers, the next step for them is to resolve the corresponding TD-related problems in the source code. To define the moment of TD remediation, we use the date of the last commit that has been determined to address the tracked issue. Consequently, *remediation latency* is the time difference between the moments of TD remediation and repayment decision.

To compare intervals, we turn to the Kruskal–Wallis test [29], which is a non-parametric method for testing whether samples originate from the same distribution. Kruskal–Wallis is used for comparing two or more independent samples of equal or different sample sizes. A significant Kruskal–Wallis test indicates that at least one sample stochastically dominates one other sample. However, the Kruskal–Wallis test does not identify where stochastic dominance occurs or for how many pairs of groups. For analyzing the specific sample pairs for stochastic dominance, we used Dunn's test [30] with Bonferroni correction.

For **RQ3**, we identify how the involvement of developers is associated with the length of the three intervals in the lifecycle, i.e., *identification latency*, *discussion period* and *remediation latency*. To this end, we first consider which developers were involved in the four moments. Especially, we create unique, anonymous ID's for each developer involved in relevant commits (i.e., introducing or paying back TD) based on their name and email.

Next, we focus on *identification latency* and use the Kaplan–Meier (K-M) method [31] to investigate whether the length of this timespan is related to whether the developer identifying the debt item (in the issue tracker) also authored a commit that introduced (part of) the debt item. The Kaplan–Meier method is a non-parametric statistic that can estimate the survival function from lifetime data. This method is also known as 'product limit estimator' and is commonly used to compare the longevity of products (in our case, TD items) under different

[12] In our dataset, TD item IDs follow the format `<issue-key>|<item-number>`. Thus, this examples refers to item #0 of the issue "*THRIFT-2768*".

Table 4
Numbers and percentages of TD items resolved in Issue Trackers (IT) are paid back in Source Code (SC).

Types	Indicator	IT	SC	%	IT	SC	%
Architecture debt	Using obsolete technology (ObsTech)	8	8	100.0	18	16	88.9
	Violation of modularity (VioMod)	10	8	80.0			
Code debt	Complex code (CpxCd)	2	0	0.0	175	154	88.0
	Dead code (DedCd)	24	20	83.3			
	Duplicated code (DupCd)	1	1	100.0			
	Low-quality code (LQualCd)	126	115	91.3			
	Multi-thread incorrectness (MTCor)	10	9	0.0			
	Slow algorithm (SlAlg)	17	14	82.4			
Design debt	Non-optimal defects (Des)	123	113	91.9	123	113	91.9
Test debt	Expensive tests (ExpTst)	3	3	100.0	47	35	74.5
	Flaky tests (FlaskTst)	11	7	63.6			
	Lack of tests (LacTst)	21	15	71.4			
	Low coverage (LCvg)	12	10	83.3			
Total					357	312	87.4

circumstances (e.g., various ways that developers are involved in the items' lifecycle).

We then look into the *discussion period* and build a generalized linear model [32] (GLM) to examine the relation between this period and two factors: the total number of developers involved in the issue tracker discussions, and whether or not the discussion involves authors of commits that introduced (part of) the debt. GLM is a multivariate analysis method that allows modeling data that is not normally distributed. In this case, the *discussion period* follows a Poisson distribution.

Finally, for *remediation latency*, we use the Kaplan–Meier method once more. In this case, we compare the remediation latency of TD items in pairs of scenarios related to kinds of developers involved in the remediation (i.e., the decision-making developer or the identifying developer).

4. Results

4.1. To what extent technical debt items identified and tagged as 'resolved' in issue trackers are actually paid back in source code? (**RQ1**)

There are 534 TD sections (i.e., summary, description and comments) involved in the selected 300 issues, of which 26 sections concern TD introduced in patches during the discussion, and then resolved in the final commits. Thus, we filtered out these TD items since they were not introduced in the source code. The remaining 508 TD sections from the selected 300 issues (no issue had all its sections removed in the previous step) can be mapped into 357 TD items. The majority of these TD items corresponded to a single TD type, and only six corresponded to more than one types.

To exemplify how multiple TD types may appear in a single item, we present one such item. In the issue HBASE-1655 "*Usability improvements to HTablePool*", the summary indicates Design Debt, but the description and the comments are related to both Design Debt and Code Debt; the latter includes the indicators Low-quality code (LQualCd) and Dead code (DedCd). From the description:

- "*Remove constructors that were not used.*" - [Code-DedCd]
- "*Change internal map from TreeMap to HashMap because I could not see any reason it needed to be sorted.*" - [Design-Des]

And from the comments of issue HBASE-1655:

- "*... The default getTable also requires instantiating a new HBaseConfiguration() each time internally, even if we are reusing an existing HTable... Whether there is a significant overhead or not to that, we should avoid it when unnecessary.*" - [Code-LQualCd]
- "*...only issues there are the tab issues, reordering, and whether to expose the pools or not.*" - [Code-LQualCd]
- "*Sorry about the tab/spaces issue. I did not clean it up carefully enough.*" - [Code-LQualCd]

Table 5
Fisher's exact test comparison of issue fixing between debt types.

Comparison			Statistic[a]
Architecture debt	x	Test debt	2.74
Architecture debt	x	Design debt	0.64
Architecture debt	x	Code debt	1.09
Code debt	x	Test debt	2.51*
Code debt	x	Design debt	0.58
Design debt	x	Test debt	4.30*

*Statistically significant (p-value < 0.05).
[a]Also represents the effect size based on odds ratio.

- "*Even though TreeMap uses the comparator rather than the equals method to compare keys, using a byte[] as the key seems to break the contract of a java.util.Map...*" - [Design-Des]
- "*...Seems better to just use String and HashMap which works well and satisfies the Map contract.*" - [Design-Des]
- "*While we lose the convenience of calling static methods...*" - [Design-Des]

Table 4 presents the results of the remaining 357 TD items (each of them only belonging to one TD indicator), including their types and indicators, and the numbers and percentages that are paid back in source code. The result shows that **most of the TD items (i.e., 87.4%) identified and mentioned as resolved in issue trackers are indeed paid back in source code**.

Regarding different types of TD, we conducted pairwise Fisher's exact tests [33] on the percentages of TD items that are paid back in source code from the different types. The results in Table 5 show that the percentage of Design Debt and Code Debt items that are also paid back in source code differs significantly from Test Debt. In particular, Test Debt items are four and two times less likely to be fixed than Design Debt and Code Debt items respectively. Altogether, we notice that **debt of all types has similarly high chance of being paid back in source code, although Test Debt's is statistically lower**.

We also compared projects and, for most cases, found no significant differences in the likelihood of issues being actually fixed in source code (see Table 6). The only exception is Hadoop (fixing ratio ≈77%), for which issues are three times and seven times less likely to be fixed in source code than issues in Camel (fixing ratio ≈93%) and Impala (fixing ratio ≈96%) respectively.

4.2. How long are the time intervals in TD items' lifecycle? (**RQ2**)

During the data analysis, we found that developers often identified a TD item from the source code and reported it in the issue tracker, but subsequently no discussion was held. Thus, it is hard or even impossible to establish when exactly developers decided to resolve

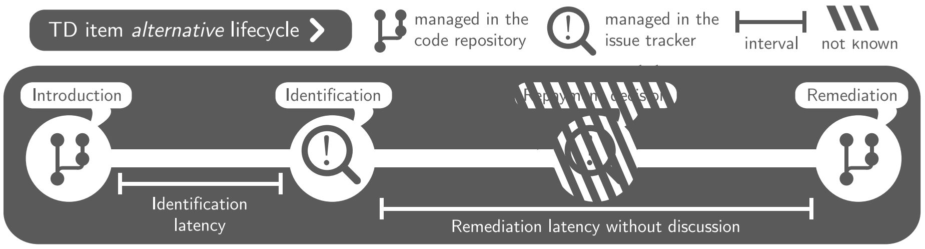

TD item alternative lifecycle
managed in the code repository
managed in the issue tracker
interval
not known
Introduction
Identification
Repayment decision
Remediation
Identification latency
Remediation latency without discussion

Fig. 3. Alternative lifecycle of TD items that manifest in both issue trackers and source code.

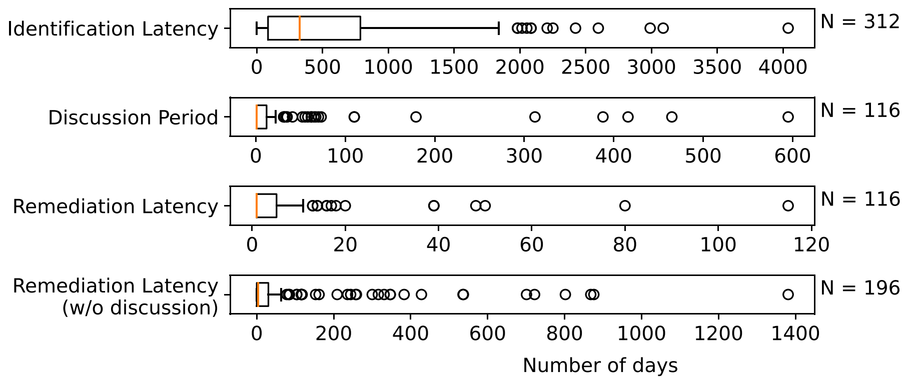


Fig. 4. Distribution of time intervals between introducing, identifying, deciding to repay and paying back all TD items.

Table 6
Fisher's exact test comparison of issue fixing between projects.

Comparison			Statistic[a]
Camel	x	Impala	0.48
Camel	x	Hadoop	3.66*
Camel	x	Hbase	1.95
Camel	x	Thrift	1.48
Impala	x	Hadoop	7.67*
Impala	x	Hbase	4.09
Impala	x	Thrift	3.10
Hadoop	x	Hbase	0.53
Hadoop	x	Thrift	0.40
Hbase	x	Thrift	0.76

*Statistically significant (p-value < 0.05).
[a]Also represents the effect size based on odds ratio.

that TD item, i.e., the moment of the repayment decision is unknown. Consequently, we consider an alternative lifecycle for those TD items with an unknown repayment decision, as illustrated in Fig. 3; *Remediation latency without discussion* refers to the time interval between the moment of TD introduction and TD remediation for those TD items. Taking into account both the original and the alternative lifecycle, we divided all identified TD items into two groups according to whether the *Repayment decision* moment could be determined or not.

Among all 312 TD items that have been resolved in source code (as shown in RQ1), the moment of the repayment decision can be confirmed for 116 of them. Taking all 312 TD items into account, Fig. 4 shows the box plots of the distribution of the number of days for intervals *identification latency*, *discussion period*, *remediation latency* and *remediation latency without discussion*. One can notice considerable differences in the descriptive statistics of the four intervals. In particular, the median values are 326, 1, 1 and 4 days for the four intervals, respectively.

To further evaluate the significance of such difference, we calculated the Kruskal–Wallis test [29] based on the number of days for the four intervals and followed it with Dunn's tests for pairwise comparisons, since we cannot assume the intervals are normally distributed. The results of the Kruskal–Wallis test (H: 334.66, p-value < 0.01, ϵ^2: 0.45) reveal that the difference between the four intervals is significant with a moderate effect size (ϵ^2). In addition, the results of the Dunn's test show that only *identification latency* has significant differences from the other three intervals (p-value < 0.01), while there is no significant difference between *Discussion period*, *Remediation latency* and *Remediation latency without discussion*. This finding indicates that **although TD items take a long time to be identified (around one year), the decision to repay them and their actual repayment in source code takes place within a few days after being identified.**

Subsequently, we performed a separate analysis related to the four considered TD types, i.e., Architecture Debt, Code Debt, Design Debt and Test Debt. Fig. 5 shows the box plots of the distribution of the number of days needed to manage the different types of TD for the four intervals. For each interval, we conducted the Kruskal–Wallis test to evaluate the significance of the difference between TD types. The results reveal no significant difference between different types of TD within the same time interval (i.e., p-value > 0.05). This finding suggests that, **while the number of days varies widely between time intervals, there is no significant difference in how developers manage different types of debt within the same time interval.**

Finally, we also investigated whether the intervals were statically different between the studied projects. We found a significant difference for the *identification latency* (H: 12.44, p-value: 0.014, ϵ^2: 0.04) and the *remediation latency without discussion* (H: 21.98, p-value < 0.01, ϵ^2: 0.11). However, the effect size in these two cases is weak and, thus, post-hoc Dunn's tests would not be meaningful. Altogether, we notice that **there is no strong difference regarding the intervals between the studied projects**.

*4.3. How are developers associated with the management of technical debt items documented in issue trackers? (**RQ3**)*

Regarding *identification latency*, we investigated whether the length of this timespan is related to whether the developer identifying the debt item (in the issue tracker) also authored a commit that introduced (part of) that item (**RQ3.1**). To answer this question, we used the Kaplan–Meier (K-M) method [31], which is a non-parametric statistic,

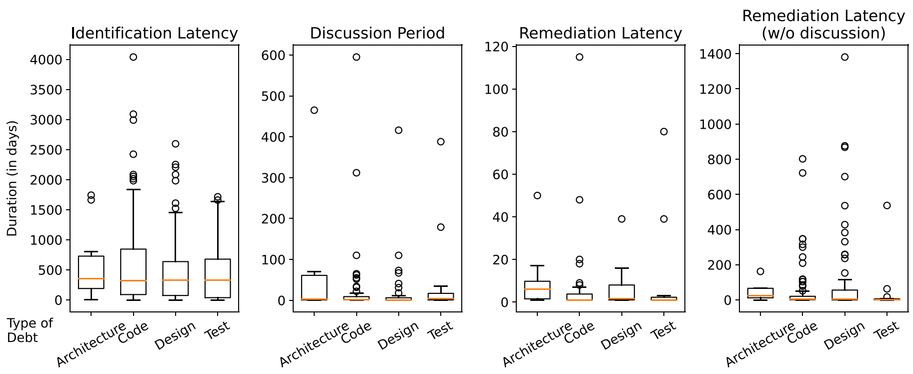


Fig. 5. Distribution of Time Intervals between Introducing, Identifying, Deciding to Repay and Paying Back Different Types of TD Items.

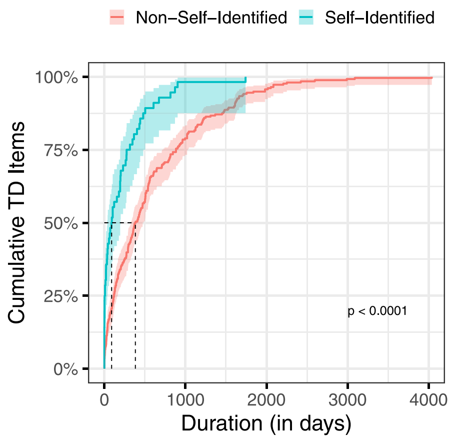


Fig. 6. Results of Kaplan–Meier method for the identification latency of all the 312 TD items.

to analyze the *identification latency* of all the 312 TD items. Fig. 6 shows the Kaplan–Meier plot depicting the cumulative percentage of TD items (y-axis) that have been identified within a specified number of days (x-axis). The red and blue areas represent the 95% confidence intervals of the survival curves.

In Fig. 6, if one item was identified and marked in the issue tracker by the same developer who introduced it in the source code, we call it a 'self-identified' item; otherwise, it is a 'non-self-identified' item. We find that there are 26 self-identified and 193 non-self-identified items from all the 312 TD items. In addition, the dotted lines in Fig. 6 show the median identification latency for both self-identified and non-self-identified TD items. From Fig. 6, we observe that self-identified items were identified in a shorter period of time than non-self-identified items. To evaluate whether such a difference is significant, we conducted a log-rank test [34] on the length of the *identification latency* of self-identified and non-self-identified items. R2.5, R2.9To estimate the effect size, we use the hazard ratio (HR) of the Cox Proportional Hazards model [35]. The log-rank test (p-value < 0.01) and hazard ratio (HR $= 2.42$)[13] shows that the difference is significant and pronounced, which indicates that **TD items are identified faster in the issue tracker by the developers who introduced them in the source code.**

Next, we focused on the *discussion period* (**RQ3.2**) and examined the relation between this period and two different factors: (a) the total number of developers involved in the issue tracker discussions (`developer_discussion`); and (b) whether or not the discussion involve developers who have also authored commit(s) that introduced

[13] HR = 1: no effect; HR < 1: reduction in the hazard; and HR > 1: increase in hazard.

Table 7
Results of the generalized linear model.

Deviance residuals					
	Min	1Q	Median	3Q	Max
	−13.186	−7.718	−5.478	−2.795	46.981
Coefficients:					
	Estimate	Std.Error	z-value	p-value	Effect size
(Intercept)	2.545	0.046	54.90	<0.001	
developers_in_discussion	0.330	0.011	30.79	<0.001	39.93%
introducer_in_discussion	−0.732	0.036	−20.49	<0.001	−51.61%

(part of) the debt (`introducer_in_discussion`). Subsequently, we built a generalized linear model [32] (GLM) to analyze whether the length of the *discussion period* correlates with these factors. Since the period data follows a Poisson distribution, we built the regression model using the Poisson family estimator.

Table 7 reports the results of the regression model; it shows that both factors have a statistically significant effect on the length of the discussion period (i.e., p-values < 0.01). From the results, we also learn that **the presence of more developers in a discussion may increase its length**, in the case of our population by an average of 40%, while **the involvement of developers who introduced the debt may reduce the length**, in this case by an average of 50%.

For *remediation latency* (**RQ3.3**), following our findings in **RQ2**, we distinguish between two cases: the regular *remediation latency* (see Fig. 1), i.e., the time difference between the moment of remediation and (known) repayment decision; and the *remediation latency without discussion* (see Fig. 3), i.e., the time interval between the moment of TD identification and TD remediation for the TD items with an unknown repayment decision. Regarding the first case (116 TD items), we investigated if the length of the timespan is related to whether the developer(s) paying back the debt in source code were also involved in the issue tracker discussion. For the second case (196 TD items), we investigated whether the length of the timespan is related to whether the developer(s) paying back the debt in source code also identified it in the issue tracker.

Starting with the first case, Fig. 7 shows the Kaplan–Meier plot to visualize the timespan curves of the 116 TD items for which the moment of the repayment decision can be confirmed. The plot depicts the cumulative percentage of TD items that have been paid back after the decision of being resolved (y-axis) within a specified number of days (x-axis). In Fig. 7, if one item was paid back in the source code by the same developer who was involved in the issue tracker discussion, we mark it as an 'involved' item; otherwise, it is a 'non-involved' item.

As shown in Fig. 7, the curves of involved and non-involved items almost overlap at the beginning: half of all items are paid back within a couple of days. Then, although around 90% of the 'involved' items are paid back within one month, the remaining ones took up to four months. The 'non-involved' items display a similar, less steep scale, but

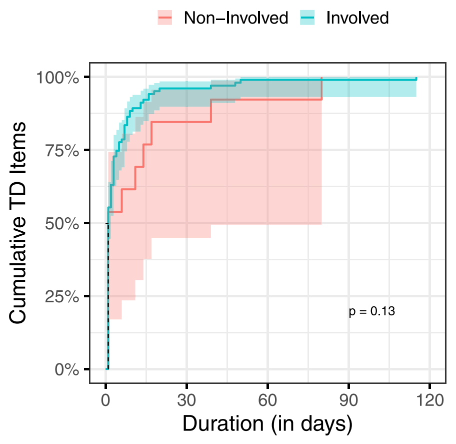


Fig. 7. Results of Kaplan–Meier method for the remediation latency of the 116 TD items.

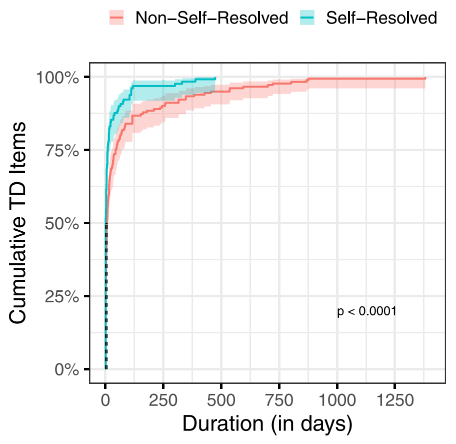


Fig. 8. Results of Kaplan–Meier method for the *remediation latency without discussion* of the 196 TD items.

are all paid back within three months. To evaluate the significance of the observed differences, we conducted a log-rank test on the length of the interval *remediation latency* of involved and non-involved items. The result shows no significant difference between the involved and non-involved TD items (p-value = 0.13). This result indicates that **whether the developer who paid back the item also participated in the issue tracker discussion has no significant effect on how long it takes to resolve the item.**

For the second case, Fig. 8 shows the Kaplan–Meier plot to visualize timespan curves of *remediation latency without discussion* for the 196 TD items, depicting the cumulative percentage of TD items that have been paid back after being identified (y-axis) within a specified number of days (x-axis). In Fig. 8, if one item was paid back in the source code by the same developer who identified it, we marked it as a 'self-resolved' item; otherwise, it is a 'non-self-resolved' item.

As shown in Fig. 8, the majority of the self-resolved items and the non-self-resolved items were fixed quickly, and the curves of those two kinds of items almost overlap at the beginning. However, the result from a log-rank test shows that the difference between the length of the *remediation latency without discussion* of self-resolved and non-self-resolved items is statistically significant (p-value < 0.0001). Moreover, the hazard ratio suggests a moderate effect (HR = 1.71).[14] Thus, **TD items seem to be resolved faster in the source code by the developers who identified them and subsequently marked them in the issue tracker**.

5. Discussion

In this section, we first elaborate on the interpretation of our results, then discuss the implications for researchers and practitioners, and finally present the threats to the validity of our study.

5.1. Interpretation of results

5.1.1. RQ1 - the likelihood of TD repayment in code

If technical debt items are identified in issue trackers and subsequently mentioned as resolved, they are most likely resolved in the source code as well; we observed this for 87.4% of our data. This means that there still remains a percentage of false positives: 12.6% of TD items marked as resolved in issues trackers are in fact not resolved in code. A related study found that only less than one third of resolved TD items that are reported in issue trackers can also be traced to source code comments [8]. The take-away here is that using a single source (e.g., code comments or issues) to investigate TD repayment may lead to false conclusions about how much TD is paid back. In contrast, combining at least two data sources, as we did in this study, allows for more certainty about the real rate of repayment for TD items reported in issue trackers. Nevertheless, we acknowledge that the identified traces were not confirmed by the developers themselves (see Section 5.3).

Furthermore, the observed rate of remediation may differ depending on the frame of reference. For example, Zabardast et al. [36] observed that 22.9% of refactoring commits paid back TD. While the starting point of our data collection are issues, Zabardast et al. [36] start from commits. Issues represent more prominent problems, as someone bothered to create a ticket for them; this might partially explain why we observed a higher fixing rate. Another relevant fact is that we could not find related commits for 34% of the issues classified as TD (in our dataset). Thus, they might as well not be fixed in the source code. However, since we cannot be sure of that, we did not consider them in our statistics.

The structure and size of the team may also play a role. We note that although the projects display similar fixing rates, we found that one project, namely Hadoop, displayed a statically significant lower fixing ratio compared to the two projects with the highest ones (i.e., Camel and Impala). We compared the characteristics of all five projects and Hadoop seems average across the board, except for two points: it is the largest project in SLOC (although not the largest SLOC/file), and it has the largest developer base. More importantly, we found that Hadoop has the lowest ratio of $\frac{\text{PMC members+Committers}}{\text{Developers}}$ (0.016), which is three times smaller than the second lowest (Hbase: 0.053) and 42 times smaller than the highest ratio (Impala: 0.685). Moreover, we found a perfect rank correlation (i.e., same order) between the fixing ratio and ratio of $\frac{\text{PMC members+Committers}}{\text{Developers}}$. Altogether, we conjecture that lack of enough oversight (in this case, people that can review code and apply changes, i.e., PMC members and Committers) impact the quality of TD management (in this case, the ratio of successfully fixed TD items).

Finally, the likelihood of the resolved TD items (in issue trackers) to be paid back in source code varies widely among different types. For example, we found that Design Debt items have a significantly higher chance to be paid back in source code, while Test Debt items are less likely. This finding is consistent with our previous study [2], where we established that Design Debt has a high fixing rate in both Python and Java projects. One possible reason might be that developers realize the impact of Design Debt on software maintenance and try to address it, to avoid paying high technical debt interest. Moreover, Design Debt seems to be a prominent concern among developers. Xavier et al. [8]

[14] HR = 1: no effect; HR < 1: reduction in the hazard; and HR > 1: increase in hazard.

found that almost 60% of the studied TD items documented in issues turned out to be related to Design Debt, and Liu et al. [37] found that Design Debt is removed the fastest in source code comments along the development process.

5.1.2. RQ2 - intervals in the lifecycle

We found out that, although TD items take a long time (around one year) to be identified in the issue tracker, they are likely to be resolved in source code within a few days after being identified. We acknowledge that there is room for variation in the measurement of these results (see Section 5.3. That said, another study that focused on investigating the remediation of TD in issue trackers also found that after TD items are reported in an issue, most of the fixes happened in a short time compared to the average (i.e., 67% of TD is repaid in the first 100 h) [4]. A possible reason could be that TD items in source code remain hidden when they do not cause trouble. As soon as they start causing trouble during maintenance (e.g. they make parts of the code harder and harder to change), developers record them as issues. Once recorded, they are visible and action is swiftly taken.

We also found no significant difference in how developers prioritize the different types of debt within the same lifecycle interval. This observation could also be explained by the reason mentioned in the previous paragraph: any maintenance issue troubling developers to the point of being reported, would likely be of similar importance to the development team.

Finally, although we observed a difference in the length of the various intervals across the studied projects, the effect size is weak, suggesting a similar treatment of issues across them. This is not surprising given that all projects follow ASF practices and contribution guidelines. Notice that this observation is not related to the efficacy of the fix (i.e., the projects' fixing rate), for which we found a difference, as explained in Section 5.1.1.

5.1.3. RQ3 - developer involvement

We found that TD items seem to be more quickly identified in the issue tracker by the same developers who introduced them in the source code. The same holds for the subsequent moments, e.g., when the same developer is involved in both identification and repayment, the corresponding TD items seen to be resolved faster. These findings are consistent with the results of our previous study [38], where we discovered that developers are often willing to address their own TD, and they identify and fix it quicker than TD incurred by others. One possible reason might be that developers are more familiar with the TD items introduced or identified by themselves. Another is the sense of responsibility that developers may have in cleaning up their own code. Nevertheless, we acknowledge that such hypotheses would require validation with developers (see Section 5.3).

However, whether developers discuss the TD items after being identified in the issue tracker seems to have no bearing on TD remediation, i.e., there is no significant effect on how long it takes to resolve the item. That said, the way developers are involved in the issue tracker discussions seems to be associated with significant changes in the length of intervals. While a larger numbers of developers participating in the discussion can increase its duration, the participation of developers who introduced the debt can reduce it. This finding shows empirical evidence of a behavior that is mainly discussed anecdotally: long-term involvement of developers in particular modules/files can support a more rapid response to the accumulation of TD beyond acceptable levels; in this case by reporting the problem in issue trackers sooner rather than later.

Altogether, the above observations seem to imply that developers, being the most knowledgeable of the software's technical issues (and debt), share the role of spearheading TD management together with the project leaders. In the same vein, only about one-third (i.e., 116/312) of TD items are discussed and then mentioned to be solved in issues; this means that the majority of TD items are acknowledged and then paid back without the apparent need to debate regarding their solution or repayment relevance. We thus conjecture that, in most cases, the expertise and knowledge of the developers affords them the authority to repay TD items without the need of technical discourse.

We note, however, that these findings may not fully translate to other contexts such as teams maintaining an industrial codebase. Although the teams investigated in this study have an established and long-standing organizational setting, these kinds of OSS structures are not often met in industry, e.g. developers vs. committers and PMC members with the right of accepting changes to the codebase. Teams in industry are more independent and most people have the right to commit and merge.

5.2. Lessons and implications

From the three research questions in this study, we have learned two main lessons. First, the most relevant technical debt items, which someone finds important enough to report as an issue, are addressed swiftly. This suggests that keeping technical debt under control is feasible if teams strive to report the harmful debt items in issue trackers. Second, the original author of a TD-incurring code does a better job of spotting issues and addressing them than other developers. Thus, keeping the original author involved in future changes of the same code, can help significantly to identify and remove TD items swiftly.

Reflecting on the results, we believe that investigating how developers manage TD across both source code repositories and issue trackers can help researchers better understand developers' actions and provide insights into how to improve the state of the practice. We have established that using two data sources for tracking the lifecycle of TD items provides a more complete picture and particularly reveals false positives on their repayment; we encourage researchers to combine two data sources or more when investigating similar topics, e.g., TD prioritization, and software maintenance workflows in general. Moreover, we found that TD items usually take a long time to be identified in issue trackers. However, once they are identified, they are likely to be resolved in the source code within a few days. Thus, the investment of research effort into developing tools to help developers identify and, more importantly, report TD items can yield noticeable improvement in debt repayment. We do not advocate flooding developers with more automatically generated information, but we emphasize the role that TD identification and prioritization can play in supporting TD management. The developers are ultimately responsible for reporting issues, and tools should be designed carefully to integrate into the developer workflow organically.

Furthermore, although our work sheds light on the dynamics of TD management, there is limited knowledge of what leads certain TD items to be discussed. Further research of such TD items may give rise to: a) methods for early detection of more challenging TD items (i.e., that require discussion); (b) the identification of recurrent solutions (e.g. design patterns or refactorings) for such items; and (c) strategies for mining expert opinions among similar TD items, which may aid their faster resolution.

Our results also offer to practitioners, a broader view of technical debt repayment in source code. First, we advocate the documentation of issue resolution in code repositories (e.g., linking issue ID in commit messages). We also encourage practitioners to report TD items found in the source code (as issues or source code comments, depending on the team's practices and preferences), even if there is no plan to address them immediately. However, we also advise to tag these 'lower-priority' TD items as such. We believe this practice would empower (automation) tools while causing little to no disturbance in the ongoing issue management practices. The explicit link between issues and resolving commits and the additional documented TD items generate traces that can provide greater analytical power to existing (and future) tools. That is, it will enable research like that mentioned in the previous paragraph.

Then researchers can feed the developed theory and methods back into practitioner tools.

Finally, our findings related to team dynamics can directly support practitioners to re-allocate suitable developers to projects. For example, we see the opportunity for a cultural change. In a previous study, we found that some developers seek to review code and identify their TD before they leave a team [38]. The results of this work show that the developers who initially incurred the debt may spot issues more easily. Thus, formalizing such an end-of-cycle activity and transform it from the 'action of individuals' into 'team actions' can bear benefits.

5.3. Threats to validity

Construct validity is related to the connection between what we intend to measure and what we are ultimately measuring. In this respect, we aimed to study TD items managed in both code repository and issue tracker. Although the data extraction process ensures that the items are managed as intended, the identification of TD items is subject to validity threats. The items are identified and classified based on textual description in the issue tracker sections. The items were not confirmed by the practitioners who documented them, but the identification/classification tasks were performed using established frameworks. Another threat concerns the survival time measured for each TD item. The measurement process (using the code repository and issue tracker) is deterministic. However, as our intention was to observe the items' three intervals, the measurement may be slightly off due to potential missing sources (i.e. sources other than source code and issue trackers). For example, a TD item may have been first identified in mailing lists or Discord/Slack channels, and only then reported in issue trackers.

External validity concerns threats to the generalizability of our findings. Given the population of our study, one cannot expect the results to apply to every development team. Potential differences in TD (and issue) management among the investigated projects and team practices (or lack thereof) can influence the introduction and survival time of TD items. We analyzed issues from five large open source projects with different programming languages and sizes, which all use Jira as the issue tracker. Although a replication with more projects may further confirm the conclusions we reached, we believe our findings can be relevant to other large, open-source projects that use Jira. To mitigate this threat, we also refrain from making conclusions in terms of order of magnitude, and stick to statistical differences between groups. Also, to avoid biases from investigating a limited sample size, we randomly selected the analyzed sample from all the collected issues. Furthermore, any tool can influence practices. Thus, a replication study that analyzes projects using other issue trackers (e.g., from GitHub) may refine some of the conclusions. Nevertheless, the main workflow and features of issue trackers are common among such tools, so we consider this threat partially mitigated.

Reliability considers the bias from the researchers in data collection or data analysis. To address these threats, two researchers were involved in the data collection and analysis. Moreover, the first and second authors classified the TD items contained in the issues independently into the different TD categories, using the description of the rules and the definition of the categories. To assess the disagreements numerically, we estimated the inter-rater agreement using Krippendorff's alpha [39] ($\alpha = 0.82$).[15] The confirmation of TD items' removal in the codebase is also subject to threats. Thus, in addition to the involvement of multiple researchers, we also describe the followed process in as much detail as possible. Finally, to support the replication of our study, we created an online repository with the necessary instructions and dataset [25], which helps researchers use and extend our dataset and replicate our analyses.

[15] Krippendorff's inspection of the tradeoffs between statistical techniques establishes that it is customary to require $\alpha \geq 0.80$ [40].

6. Related work

In this paper, we combine two data sources (i.e., source code repository and issue tracker) to investigate the lifecycle of TD items (introduction, identification, discussion and repayment), and how developer involvement influences this lifecycle. In this section we discuss the state of the art regarding TD management based on one or multiple data sources, and the influence of developer-related factors on TD management.

6.1. TD management based on one or more sources

Most of the current studies focused on investigating TD introduction and remediation based on source code artifacts. In this context, several relevant findings have been reported. First, a large percentage of TD is indeed paid back during software evolution [41–43]. With the evolution of Java and Python projects, the number of TD items shows a significant growth trend, while a large number of TD is paid back [2,44]. Moreover, most TD items are paid back within one year of being introduced by the developers, and only a few can remain in the systems for several years [2,45,46].

More recently, a few studies focused on investigating the identification and repayment of TD based on issue tracker artifacts. For example, Bellomo et al. [19] and Dai et al. [47] analyzed the summary and description of issues in multiple trackers and found that between 4% and 9% of them discussed TD, indicating that developers indeed declare and discuss TD in the issue trackers. However, these studies did not include detailed issue information (e.g., discussion messages) and focused on identification. Subsequently, Li et al. [4] investigated the identification and remediation of TD (in particular, self-admitted technical debt or SATD) in 500 issues from two large open source projects and found that on average, 71.7% of identified TD is paid back.

The aforementioned studies that analyzed TD using a single data source have limitations. For example, Tufano et al. [46] suggest that the main reason for SATD removal is because the code is simply no longer there, instead of developers intentionally managing them. Zampetti et al. [6] looked into the question of removal "by accident" and found that 20%–50% of SATD comments are accidentally removed when entire classes or methods are dropped. Another study analyzed commit messages of Apache Java projects and found that empty commit messages are potential indicators of TD, while detailed commit messages have a negative association with the presence of TD [48].

The investigation of multiple data sources for TD research is still in its infancy. The most notorious example is a study by Xavier et al. [8], who focused on the payment of SATD items in issue tracker systems (referred to as SATD-I) and checked whether they were also documented in code comments. The results show that only 29% of the studied SATD-I items can be tracked to source code comments. Xavier et al. also found that most developers paid SATD-I to reduce interest and promote clean code. However, there is still a lack of research investigating TD repayment based on both issue tracker and source code.

Compared to this work, our study advances the state of the art by: (a) combining two data sources to investigate the entire lifecycle of TD items (i.e., introduction, reporting, discussion and remediation); and (b) performing a more in-depth analysis of issue tracker information (i.e., including summaries, description, and discussion comments), also tailored to allow the identification of TD remediation and the link between source code and issue tracker artifacts.

6.2. Developer-related factors and TD management

Technical debt management is an essential part of software development and maintenance, and commonly involves the collaboration of multiple project team members, especially for large software systems. However, some factors related to the development team can have an impact on software quality. For example, a strong sense of responsibility among developers helps to deliver higher quality code with fewer defects [49]. Currently, only a few studies have investigated the relationship between developer-related factors and TD [50,51]. Such work can further explain the reasons why developers introduce or pay back technical debt in different ways.

In our previous work, we had introduced the concept of self-fixed technical debt [52], investigating how TD is managed when developers pay back the debt that they introduced themselves. Our results show that: (a) most TD items in the source code are eventually paid back by the developers who introduced them; and (b) self-fixed TD has a shorter survival time. Some studies have also found that the majority of the SATD removals are made by the same developers who introduced them [5,9] and that TD in issue trackers is also likely to be paid back by TD identifiers and creators [4]. These studies suggest that the phenomenon of self-fixed TD is prevalent across different data sources and that self-fixing may influence the lifecycle of TD items.

Since developers may have different motives to pay back TD, we also explored the impact of some developer-related factors on the extent to which TD is paid back [53]. The findings show that the more developers maintain the same software module, the lower the chance of technical debt being self-fixed in that module, and that developers who contribute more in the software development process are more likely to repay their technical debt. These may be due to the fact that developers who are more involved in the software development process can accumulate more experience and have a better understanding of the source code. Such an explanation is consistent with the findings of Alfayez et al. [51] and Amanatidis et al. [50] that developers with more experience in software projects have a lower chance of introducing TD in the source code. However, developers with more experience tend to identify more SATD in code comments [23]. This difference may be due to the nature of SATD, i.e., more experienced developers find it easier to identify TD and, thus, document it in the code comments.

In summary, the aforementioned studies suggest that developer-related factors may influence decisions to introduce and pay back TD. However, it is not clear to what extent such factors play a role throughout the complete lifecycle of TD items. This paper addresses this aspect by focusing on the developers' involvement between TD introduction, identification, repayment decision and remediation.

7. Conclusions

This paper reports on an case study that investigated the lifecycle of technical debt items resolved in issue trackers. Specifically, we investigate the likelihood of TD items being also resolved in source code, the length between the various moments in the lifecycle, and how the number and involvement of developers are related to the lifecycle. We randomly selected 600 issues from each project, totaling 3000 issues from the five projects. Then, we manually analyzed 300 issues that contained technical debt items and used *git log* to extract the related code commits. Finally, we extracted 357 TD items from the analyzed 300 issues.

We found that most of the TD items identified and mentioned as resolved in issue trackers (i.e., 87.4%) are also paid back in source code, while about one-third (i.e., 116/312) are further discussed after being identified in the issue tracker. By looking at the timespan between the various moments in the lifecycle of a TD item, results showed that items stay dormant for most of their existence and are often resolved quickly (within a few days) after identification in the issue tracker.

Regarding the different types of TD, items of all types have a similarly high chance of being paid back in source code, although Test Debt's is lower to some degree but statistically significant. Still, we found no significant difference in how developers prioritize the different types of debt within the same lifecycle moment.

Finally, TD items seem to be reported faster when the developer who introduced them is the reporter; they are also resolved quicker when the developer who identified the items is the committer. Furthermore, the presence of more developers in a discussion may increase its duration, while the presence of developers who introduced the item in the issue tracker discussions can reduce the duration.

CRediT authorship contribution statement

Jie Tan: Conceptualization, Methodology, Software, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Daniel Feitosa:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration. **Paris Avgeriou:** Conceptualization, Validation, Writing – original draft, Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

One or more of the authors of this paper have disclosed potential or pertinent conflicts of interest, which may include receipt of payment, either direct or indirect, institutional support, or association with an entity in the biomedical field which may be perceived to have potential conflict of interest with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2023.107216. Paris Avgeriou reports financial support was provided by ITEA3 and RVO.

Data availability

We have shared the link to replication package in the paper

Acknowledgments

This work was supported by ITEA3 and RVO under grant agreement No. 17038 VISDOM (https://visdom-project.github.io/website/). Also, we would like to thank João Paulo Biazotto for his invaluable support in the data collection.

References

[1] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, J. Syst. Softw. 101 (C) (2015) 193–220.

[2] J. Tan, D. Feitosa, P. Avgeriou, M. Lungu, Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem, J. Softw.: Evol. Process 33 (4) (2021) e2319.

[3] G. Digkas, A. Chatzigeorgiou, A. Ampatzoglou, P. Avgeriou, Can clean new code reduce technical debt density? IEEE Trans. Softw. Eng. 48 (5) (2022) 1705–1721, http://dx.doi.org/10.1109/tse.2020.3032557.

[4] Y. Li, M. Soliman, P. Avgeriou, Identification and remediation of self-admitted technical debt in issue trackers, in: Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '20), IEEE, Portorož, Slovenia, 2020, pp. 495–503.

[5] E.D.S. Maldonado, R. Abdalkareem, E. Shihab, A. Serebrenik, An empirical study on the removal of self-admitted technical debt, in: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME '17), IEEE, Shanghai, China, 2017, pp. 238–248, http://dx.doi.org/10.1109/icsme.2017.8.

[6] F. Zampetti, A. Serebrenik, M. Di Penta, Was self-admitted technical debt removal a real removal? An in-depth perspective, in: Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18), ACM, Gothenburg, Sweden, 2018, pp. 526–536.

[7] R. Maipradit, C. Treude, H. Hata, K. Matsumoto, Wait for it: identifying "On-Hold" self-admitted technical debt, Empir. Softw. Eng. 25 (5) (2020) 3770–3798, http://dx.doi.org/10.1007/s10664-020-09854-3.

[8] L. Xavier, F. Ferreira, R. Brito, M.T. Valente, Beyond the code: Mining self-admitted technical debt in issue tracker systems, in: Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20), 2020, pp. 137–146.

[9] G. Bavota, B. Russo, A large-scale empirical study on self-admitted technical debt, in: Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR '16), IEEE, Austin, TX, USA, 2016, pp. 315–326.

[10] N. Ramasubbu, C.F. Kemerer, Managing technical debt in enterprise software packages, IEEE Trans. Softw. Eng. 40 (8) (2014) 758–772, http://dx.doi.org/10.1109/TSE.2014.2327027.

[11] N.S. Alves, T.S. Mendes, M.G. de Mendonça, R.O. Spínola, F. Shull, C. Seaman, Identification and management of technical debt: A systematic mapping study, Inf. Softw. Technol. (2016) 100–121, http://dx.doi.org/10.1016/j.infsof.2015.10.008.

[12] M. Vidoni, Z. Codabux, F.H. Fard, Infinite technical debt, J. Syst. Softw. 190 (2022) 111336, http://dx.doi.org/10.1016/j.jss.2022.111336.

[13] M. Wiese, P. Rachow, M. Riebisch, J. Schwarze, Preventing technical debt with the TAP framework for Technical Debt Aware Management, Inf. Softw. Technol. 148 (2022) 106926, http://dx.doi.org/10.1016/j.infsof.2022.106926.

[14] A. Ampatzoglou, A. Chatzigeorgiou, E.M. Arvanitou, S. Bibi, SDK4ED: A platform for technical debt management, Softw. - Pract. Exp. 52 (8) (2022) 1879–1902, http://dx.doi.org/10.1002/spe.3093.

[15] R. Maipradit, B. Lin, C. Nagy, G. Bavota, M. Lanza, H. Hata, K. Matsumoto, Automated identification of on-hold self-admitted technical debt, in: Proceedings of the IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM '20), 2020, pp. 54–64, http://dx.doi.org/10.1109/SCAM51674.2020.00011.

[16] D. Tsoukalas, N. Mittas, A. Chatzigeorgiou, D. Kehagias, A. Ampatzoglou, T. Amanatidis, L. Angelis, Machine learning for technical debt identification, IEEE Trans. Softw. Eng. 48 (12) (2022) 4892–4906, http://dx.doi.org/10.1109/TSE.2021.3129355.

[17] Y. Li, M. Soliman, P. Avgeriou, L. Somers, Self-admitted technical debt in the embedded systems industry: An exploratory case study, IEEE Trans. Softw. Eng. (2022) 1–22, http://dx.doi.org/10.1109/tse.2022.3224378.

[18] I. Griffith, H. Taffahi, C. Izurieta, D. Claudio, A simulation study of practical methods for technical debt management in agile software development, in: Proceedings of the 2014 Winter Simulation Conference, IEEE, Savannah, GA, USA, 2014, pp. 1014–1025, http://dx.doi.org/10.1109/wsc.2014.7019961.

[19] S. Bellomo, R.L. Nord, I. Ozkaya, M. Popeck, Got technical debt? Surfacing elusive technical debt in issue trackers, in: Proceedings of the 13th IEEE/ACM Working Conference on Mining Software Repositories (MSR '16), IEEE, Austin, TX, USA, 2016, pp. 327–338.

[20] C. Wohlin, A. Aurum, Towards a decision-making structure for selecting a research design in empirical software engineering, Empir. Softw. Eng. 20 (6) (2014) 1427–1455, http://dx.doi.org/10.1007/s10664-014-9319-7.

[21] P. Runeson, M. Host, A. Rainer, B. Regnell, Case Study Research in Software Engineering: Guidelines and Examples, Wiley Blackwell, 2012.

[22] R. Van Solingen, V. Basili, G. Caldiera, H.D. Rombach, Goal question metric (GQM) approach, in: Encyclopedia of Software Engineering, John Wiley & Sons, Inc., Hoboken, NJ, USA, 2002, pp. 528–532.

[23] A. Potdar, E. Shihab, An exploratory study on self-admitted technical debt, in: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME '14), IEEE, Victoria, British Columbia, Canada, 2014, pp. 91–100.

[24] N. Alves, L.F. Ribeiro, V. Caires, T. Mendes, R. Spínola, Towards an ontology of terms on technical debt, in: Proceedings of the 6th International Workshop on Managing Technical Debt (MTD '14), IEEE, Victoria, BC, Canada, 2014, pp. 1–7.

[25] J. Tan, D. Feitosa, P. Avgeriou, Replication package for "The Lifecycle of Technical Debt that Manifests in both Source Code and Issue Trackers", Available online at https://doi.org/10.5281/zenodo.7810853.

[26] M. Fischer, M. Pinzger, H. Gall, Populating a release history database from version control and bug tracking systems, in: Proceeding of the 19th International Conference on Software Maintenance (ICSM '03), IEEE, Amsterdam, the Netherlands, 2003, pp. 23–32.

[27] F. Shull, J. Singer, D.I. Sjøberg, Guide to Advanced Empirical Software Engineering, Springer, 2007.

[28] W.G. Cochran, Sampling Techniques, John Wiley & Sons, 2007.

[29] N. Breslow, A generalized Kruskal-Wallis test for comparing K samples subject to unequal patterns of censorship, Biometrika 57 (3) (1970) 579–594.

[30] O.J. Dunn, Multiple comparisons using rank sums, Technometrics 6 (3) (1964) 241–252.

[31] E.L. Kaplan, P. Meier, Nonparametric estimation from incomplete observations, J. Amer. Statist. Assoc. 53 (282) (1958) 457–481.

[32] J.A. Nelder, R.W.M. Wedderburn, Generalized linear models, J. R. Stat. Soc. A 135 (3) (1972) 370, http://dx.doi.org/10.2307/2344614.

[33] R.A. Fisher, On the interpretation of χ 2 from contingency tables, and the calculation of P, J. R. Stat. Soc. 85 (1) (1922) 87–94.

[34] J.M. Bland, D.G. Altman, The logrank test, BMJ 328 (7447) (2004) 1073, http://dx.doi.org/10.1136/bmj.328.7447.1073.

[35] D.R. Cox, Regression models and life-tables, J. R. Stat. Soc. Ser. B Stat. Methodol. 34 (2) (1972) 187–202, http://dx.doi.org/10.1111/j.2517-6161.1972.tb00899.x.

[36] E. Zabardast, J. Gonzalez-Huerta, D. Smite, Refactoring, bug fixing, and new development effect on technical debt: An industrial case study, in: Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '20), IEEE, 2020, pp. 376–384, http://dx.doi.org/10.1109/seaa51224.2020.00068.

[37] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, S. Li, An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks, Empir. Softw. Eng. 26 (2) (2021) 1–36.

[38] J. Tan, D. Feitosa, P. Avgeriou, Do practitioners intentionally self-fix Technical Debt and why? in: Proceedings of the 37th IEEE International Conference on Software Maintenance and Evolution (ICSME '21), 2021, pp. 251–262, http://dx.doi.org/10.1109/ICSME52107.2021.00029.

[39] K. Krippendorff, Computing Krippendorff's alpha-reliability, in: Departmental Papers (ASC), 2011, pp. 1–12.

[40] K. Krippendorff, Content Analysis: An Introduction to its Methodology, Sage publications, 2018.

[41] V. Lenarduzzi, N. Saarimaki, D. Taibi, On the diffuseness of code technical debt in java projects of the apache ecosystem, in: Proceedings of the IEEE/ACM 2rd International Conference on Technical Debt (TechDebt '19), IEEE, Montreal, QC, Canada, 2019, pp. 98–107.

[42] M.T. Baldassarre, V. Lenarduzzi, S. Romano, N. Saarimäki, On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube, Inf. Softw. Technol. 128 (2020).

[43] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC '10), IEEE, Porto, Portugal, 2010, pp. 106–115, http://dx.doi.org/10.1109/QUATIC.2010.16.

[44] G. Digkas, M. Lungu, A. Chatzigeorgiou, P. Avgeriou, The evolution of technical debt in the apache ecosystem, in: Proceedings of the 11th European Conference on Software Architecture (ECSA '17), Springer, Canterbury, UK, 2017, pp. 51–66.

[45] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, A. Ampatzoglou, How do developers fix issues and pay back technical debt in the apache ecosystem? in: Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18), IEEE, Campobasso, Italy, 2018, pp. 153–163.

[46] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M.D. Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Trans. Softw. Eng. 43 (11) (2017) 1063–1088, http://dx.doi.org/10.1109/TSE.2017.2653105.

[47] K. Dai, P.B. Kruchten, Detecting technical debt through issue trackers, in: Proceeing of the 5th International Workshop on Quantitative Approaches to Software Quality (QuASoQ '17), 2017, pp. 59–65.

[48] C. Lu, But do commit messages matter? An empirical association analysis with technical debt, in: Joint Proceedings of the Summer School on Software Maintenance and Evolution, Tampere, Finland, 2019, pp. 45–53.

[49] F. Rahman, P. Devanbu, Ownership, experience and defects: a fine-grained study of authorship, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), IEEE, Honolulu, HI, USA, 2011, pp. 491–500, http://dx.doi.org/10.1145/1985793.1985860.

[50] T. Amanatidis, A. Chatzigeorgiou, A. Ampatzoglou, I. Stamelos, Who is producing more technical debt? A personalized assessment of TD principal, in: Proceedings of the XP2017 Scientific Workshops, ACM, Cologne, Germany, 2017, pp. 1–8, http://dx.doi.org/10.1145/3120459.3120464.

[51] R. Alfayez, P. Behnamghader, K. Srisopha, B. Boehm, An exploratory study on the influence of developers in technical debt, in: Proceedings of the First IEEE/ACM International Conference on Technical Debt (TechDebt '18), IEEE, Gothenburg, Sweden, 2018, pp. 1–10.

[52] J. Tan, D. Feitosa, P. Avgeriou, An empirical study on self-fixed technical debt, in: Proceedings of the 3rd IEEE/ACM International Conference on Technical Debt (TechDebt '20), ACM, 2020, pp. 1–12, http://dx.doi.org/10.1145/3387906.3388621.

[53] J. Tan, D. Feitosa, P. Avgeriou, Does it matter who pays back Technical Debt? An empirical study of self-fixed TD, Inf. Softw. Technol. 143 (2022) 106738, http://dx.doi.org/10.1016/j.infsof.2021.106738.